91602
new 82-10

ontario hydro
research division

91-04558/

DISCHARGE ALLOCATION FOR HYDRO-ELECTRIC GENERATING STATIONS

Report No: 90-170-K

E.M. Klein
Engineer
Operations & Decision Analysis Section
Operations Research Department

ABSTRACT

This study is concerned with the allocation of discharge among generating units of a hydro-electric generating station so as to maximize power output. A hybrid separable programming-heuristic algorithm has been developed for this discharge allocation problem. It was written in C and has been implemented on an IBM-compatible personal computer. This algorithm was tested and found to increase the energy production of the R.H. Saunders Hydraulic Generating Station in excess of 12,000 MWh for a one-year period.

job	file	date	report no.
71-001-206	809.306	November 23, 1990	90-170-K

TABLE OF CONTENTS

		Page
1. INTRODUCTION		1
2. MODEL FORMULATION		1
2.1 Definitions		1
2.2 Formulation		2
2.3 Solution Approach		2
3. NUMERICAL RESULTS		4
4. RECOMMENDATIONS AND CONCLUSIONS		7
REFERENCES		7
APPENDIX 1:	Separable Programming	8
APPENDIX 2:	Example	9
APPENDIX 3:	User Guide	10
APPENDIX 4:	Function Definitions	12
APPENDIX 5:	Flow Diagram	14
APPENDIX 6:	Heuristics	15
APPENDIX 7:	Variable Definitions	16
APPENDIX 8:	HYDOPT Listing	20
DISTRIBUTION		-last page-

91601
new 82-10

ontario hydro
research division

To: Mr. A.M. Wilson
Manager
Hydraulic Stations Department
Hydraulic Generation & Transmission Operations Division

DISCHARGE ALLOCATION FOR HYDRO-ELECTRIC GENERATING STATIONS

1. INTRODUCTION

The power produced by an individual hydro-electric generating unit is a function of the head and the discharge of water through that particular unit. The power output of a multi-unit hydro-electric generating station is a function of the head and the discharge allocation among its generating units. If the discharge is not properly allocated among the generating units, the potential electrical energy may not be fully achieved. Any loss would have to be replaced by more expensive sources such as nuclear, coal or oil which are environmentally less attractive. Therefore, it is in the best interest of Ontario Hydro to maximize hydraulic generation which is the cheapest and cleanest source of energy.

The problem of allocating discharge to maximize power output has been previously studied by Bernholtz/2/ and Sim/4/. This project is a continuation of the preliminary study performed by the latter author. A computer algorithm called HYDOPT has been implemented in C using separable programming and two heuristics. This program has the flexibility for adaptation to any hydro-electric generating station configuration. The model formulation, results and conclusions will be presented in the following sections.

2. MODEL FORMULATION

In this section, a mathematical programming model is developed for allocating discharge among generating units to maximize the power output.

2.1 Definitions

n total number of generating units in the hydro-electric generating station.

i generating unit number, i = 1,2,...,n.

x_i discharge through generating unit i.

M_i maximum discharge allowed through generating unit i.

E_i discharge through generating unit i which will result in maximum efficiency of the unit.

L_i minimum discharge through generating unit i if $x_i > 0$.

K_i discharge through generating unit i below which no power is produced.

$P_i(x_i)$ power output resulting from a discharge x_i through generating unit i for a fixed head.

Q total discharge through the station.

2.2 Formulation

The model is formulated as follows:

Maximize $P_i(x_i)$

subject to $x_i \leq M_i$ $\quad i=1,...,n \quad (1)$

$$\sum_{i=1}^{n} x_i = Q \quad (2)$$

$x_i \geq 0$ $\quad i=1,...,n \quad (3)$

The model determines the discharge allocation for a fixed head to maximize the power output. Therefore, the objective function in the model is only a function of discharge allocation. Figure 1 shows a typical power curve which characterizes a hydro-electric generating unit for a specific head. The power curve is nonsmooth, nonconcave and nonlinear. A certain quantity of water (K_i) must be discharged through a generating unit before any power is produced. Point E_i represents the maximum efficiency point on the power curve. If any water is to be discharged through the generating unit, point L_i specifies the minimum amount. Any discharge of water less than L_i causes vibration problems. In practice, operating a generating unit below L_i is inefficient. It is unlikely that HYDOPT would discharge less water than L_i. For this reason, this constraint is not explicitly stated in the model formulation. All the constraints in the model are linear. Constraint (1) ensures that each generating unit will not discharge more water than is possible. Constraint (2) specifies that the total outflow of the individual generating units must equal the total inflow into the generating station. The last constraint (3) ensures non-negativity.

2.3 Solution Approach

Separable programming is a mathematical programming technique that uses piecewise linear approximations of the nonlinear functions. The power curve for each hydro-electric generating unit is approximated by 20 points. Adjacent points are connected by straight lines. In the separable programming solution, at most 2 points are allowed to approximate a point on the power curve and they must be adjacent. If the points are not adjacent, the accuracy of the approximation will deteriorate. Once the problem has been formulated as a

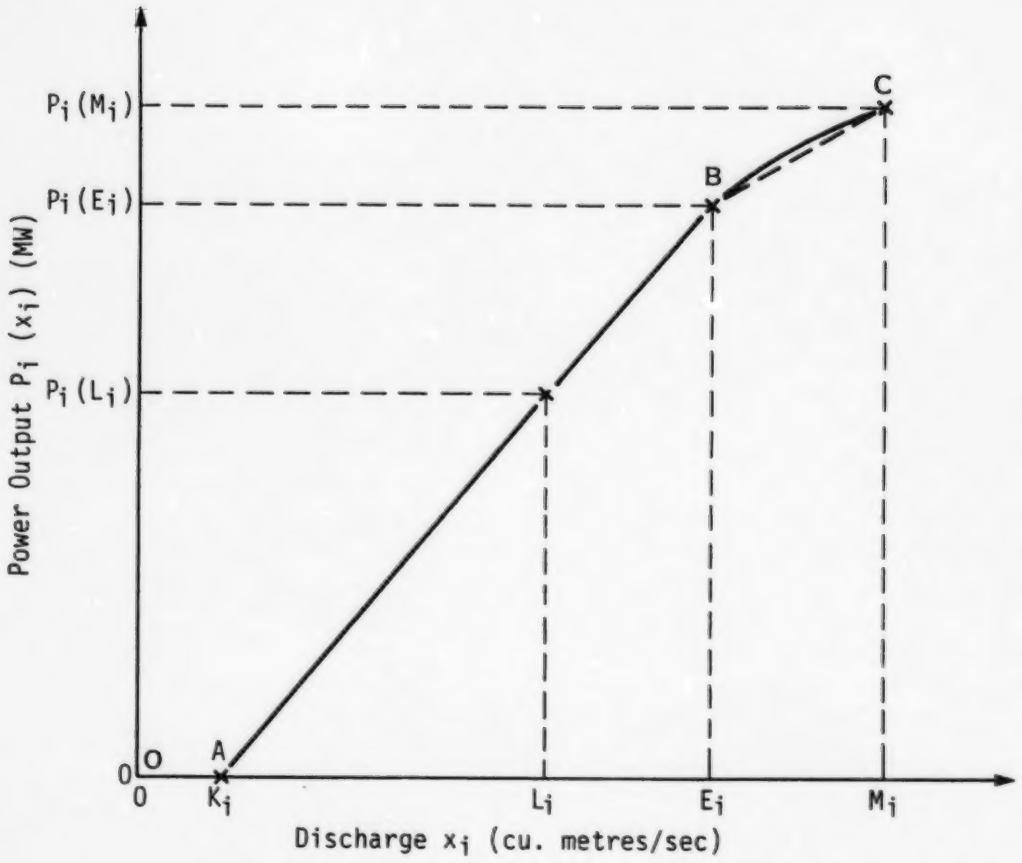

FIGURE 1

TYPICAL OUTPUT-DISCHARGE CURVE
FOR A HYDRAULIC GENERATING UNIT

separable programming problem, it is solved as a linear programming problem with the adjacency restrictions in place. Appendix 1 gives a detailed description of the separable programming formulation. Bazaraa and Shetty/1/ provide the mathematical development of separable programming. Appendix 2 shows the formulation and solution of a small hydro-electric generating problem using separable programming.

Once HYDOPT has solved the separable programming problem, the adjacency restrictions on the points are relaxed. The separable programming problem is then solved as a pure linear programming problem.

HYDOPT then performs two heuristics so as to increase the power output. The first heuristic is performed after the separable programming is completed. Water is reallocated between generating units (that are currently discharging water) if the reallocated water produces more power flowing through another generating unit. The reallocation process is based on the slopes of the piecewise linear approximation of the power curves. The first heuristic is also performed after the separable programming problem is solved as a pure linear programming problem.

The second heuristic is performed following the first heuristic. The generating unit with the nonadjacent points is shutdown. The water from the shutdown generating unit is reallocated to the other generating units that are discharging water. This heuristic is also based on the slopes of the piecewise-linear approximation. For a more detailed description of the algorithm, see Appendices 4, 5 and 6 which contain definitions of the functions used, a flow diagram of the algorithm and a description of the two heuristics.

3. NUMERICAL RESULTS

The HYDOPT test runs were performed by Sharp/3/ of the Hydraulic Stations Department on the R.H. Saunders Hydraulic Generating Station. The performance of HYDOPT is compared to the historical energy production for the period from June 1, 1988 to May 31, 1989 inclusive. This analysis is done on an hourly basis. The head, flow, energy produced and unit availability were recorded on log sheets by the operators. HYDOPT requires the head and flow to be entered as inputs. The resultant outputs are the unit allocation information and the energy production. This enables us to directly compare the historical energy production to the energy production based on HYDOPT. Due to the vagueness of some of the unit availability information, any unit not operating for a 24-hour period is assumed to be unavailable. This will provide a conservative estimate of the additional energy production if HYDOPT had been used.

Table 1 shows the additional daily energy production from HYDOPT for the month of June, 1988. HYDOPT produces additional energy for every day of June, 1988. Table 2 summarizes the additional monthly energy production from HYDOPT for June, 1988, to May, 1989. HYDOPT produces an additional 12,163 MWh of energy for the 12-month period tested. This represents a 0.194% increase over the historical energy production at the R.H. Saunders Hydraulic Generating Station.

The period from January 1989 to December 1989 is considered to be more representative of the future energy production at the R.H. Saunders Hydraulic Generating Station taking into account the station rehabilitation and maintenance plan for the next 6 to 10 years. The energy production for this period was 6,443,742 MWh. Applying the 0.194% increase in energy production leads to an estimated additional 12,500 MWh/year. Table 3 summarizes the benefits from the application of the HYDOPT algorithm. The projected savings are $437,500 for 1991 and rise to $550,000 in 1995.

TABLE 1

Comparison of Actual Energy Production Versus HYDOPT-Based Energy Production at R.H. Saunders Hydraulic Generating Station

Day	Historical Energy Production (MWh)	HYDOPT-Based Energy Production (MWh)	Additional Daily Energy Production from HYDOPT versus Historical (MWh)
June 1/88	18957	18983	+ 26
June 2/88	18897	18925	+ 28
June 3/88	18803	18857	+ 54
June 4/88	18848	18877	+ 29
June 5/88	19001	19048	+ 47
June 6/88	18971	18997	+ 26
June 7/88	18909	18974	+ 65
June 8/88	18832	18869	+ 37
June 9/88	18827	18860	+ 33
June 10/88	18804	18862	+ 58
June 11/88	18711	18762	+ 51
June 12/88	18733	18768	+ 35
June 13/88	18722	18788	+ 66
June 14/88	18578	18648	+ 70
June 15/88	18666	18732	+ 66
June 16/88	18651	18717	+ 66
June 17/88	18547	18595	+ 48
June 18/88	18399	18427	+ 28
June 19/88	18450	18481	+ 31
June 20/88	18478	18497	+ 19
June 21/88	18325	18495	+ 170
June 22/88	18468	18496	+ 28
June 23/88	18464	18482	+ 18
June 24/88	18428	18448	+ 20
June 25/88	18304	18325	+ 21
June 26/88	18402	18416	+ 14
June 27/88	18361	18365	+ 4
June 28/88	18353	18394	+ 41
June 29/88	18380	18392	+ 12
June 30/88	18317	18336	+ 19
Total	**558586**	**559816**	**+1230**

TABLE 2

Comparison of Actual Energy Production Versus HYDOPT-Based Energy Production at R.H. Saunders Hydraulic Generating Station

Month	Historical Energy Production (MWh)	HYDOPT-Based Energy Production (MWh)	Additional Monthly Energy Production From HYDOPT Versus Historical (MWh)
June 1988	558586	559816	+ 1230
July 1988	559245	560028	+ 783
August 1988	560994	561646	+ 652
September 1988	533969	534749	+ 780
October 1988	541567	542303	+ 736
November 1988	537963	539577	+ 1614
December 1988	552842	554148	+ 1306
January 1989	485259	485871	+ 612
February 1989	450208	451120	+ 912
March 1989	489871	491027	+ 1156
April 1989	468517	469695	+ 1178
May 1989	533767	534971	+ 1204
12-Month Total	**6272788**	**6284951**	**+12163**

TABLE 3

Benefits Resulting from Application of HYDOPT Algorithm

Year	Estimated Hydraulic Energy Replacement Cost ($/MWh)/5/	Saving Due to HYDOPT Algorithm ($/Year)
1991	35	437,500
1992	38	475,000
1993	38	475,000
1994	40	500,000
1995	44	550,000

4. RECOMMENDATIONS AND CONCLUSIONS

HYDOPT produces additional energy in comparison with the historical energy production at the R.H. Saunders Hydraulic Generating Station for the period from June, 1988 to May, 1989. It is estimated that an additional 12,500 MWh/year can be achieved with the implementation of the HYDOPT algorithm. This reduces costs by replacing electrical energy generated from oil, fossil, and nuclear sources. The projected savings for 1991 are $437,500 and will rise to $550,000 in 1995. Since hydraulic energy is Ontario Hydro's cleanest and least expensive form of energy, it should be fully utilized before resorting to thermal energy sources which emit acid and carbon-based gases harmful to the environment.

The HYDOPT algorithm should be implemented at the R.H. Saunders Hydraulic Generating Station. This will provide an opportunity for observing the performance of HYDOPT under actual operating conditions. It will also allow station operators to provide useful comments on the HYDOPT program.

Since HYDOPT is a general optimization algorithm, it can be implemented at other hydro-electric generating stations.

Approved:

A.H. Chung
Manager
Operations Research Department

Submitted:

Erich M. Klein

E.M. Klein
Engineer
Operations & Decision Analysis Section

EMK:cmh

REFERENCES

1. Bazaraa, M.S., and Shetty, C.M., "Nonlinear Programming Theory and Algorithms", John Wiley and Sons, New York, 1979.

2. Bernholtz, B., "Optimum Allocation of Discharge to Units in a Hydro-Electric Generating Station", SIAM Review, 2, 1969.

3. Sharp, B., "Projected Benefits of Optimal Flow Dispatch at R.H. Saunders G.S.", Technical Report, Hydraulic Stations Department, Ontario Hydro, October 16, 1990.

4. Sim, S.H., "Discharge Allocation of a Hydro-Electric Generating Station - A Preliminary Study", Ontario Hydro Research Division Report 87-92-K, 1987.

5. Production Branch Integration Plan, Ontario Hydro, March 1990.

APPENDIX 1

Separable Programming

n = total # of generating units in the hydro-electric generating station.

m = # of grid points per power curve.

x_{ij} = flow through generating unit i at flow point j.

λ_{ij} = grid point variable of power curve i at flow point j.

$P_i(x_{ij})$ = power produced by generating unit i at flow point j.

M_i = maximum quantity of discharge allowed through generating unit i.

Q = total discharge through the station.

$$\text{Maximize} \sum_{i=1}^{n} \sum_{j=1}^{m} P_i(x_{ij})\ \lambda_{ij}$$

$$\text{Subject to} \sum_{j=1}^{m} x_{ij}\ \lambda_{ij} \leq M_i \qquad i=1,\ldots, n \qquad (1)$$

$$\sum_{i=1}^{n} \sum_{j=1}^{m} x_{ij}\ \lambda_{ij} = Q \qquad (2)$$

$$\sum_{j=1}^{m} \lambda_{ij} = 1 \qquad i=1,\ldots, n \qquad (3)$$

$$x_{ij},\ \lambda_{ij} \geq 0, \text{ for all } i,j \qquad (4)$$

For each i, at most 2 nonzero adjacent λ_{ij}'s exist.

APPENDIX 2

Example

Consider a hydro-electric generation station with two generating units. A power curve with 5 grid points will be used. 525 m^3/s will be discharged.

Generating Unit 1		Generating Unit 2	
flow (m^3/s)	power (MW)	flow (m^3/s)	power (MW)
0	0	0	0
71.9	0	78.4	0
217.5	45	244.0	50
242.6	52	268.5	57
304.0	58	321.2	63

Maximize $45\,\lambda_{13} + 52\,\lambda_{14} + 58\,\lambda_{15}$

$+ 50\,\lambda_{23} + 57\,\lambda_{24} + 63\,\lambda_{25}$

Subject to $71.9\,\lambda_{12} + 217.5\,\lambda_{13} + 242.6\,\lambda_{14} + 304.0\,\lambda_{15} \leq 304.0$

$78.4\,\lambda_{22} + 244.0\,\lambda_{23} + 268.5\,\lambda_{24} + 321.2\,\lambda_{25} \leq 321.2$

$71.9\,\lambda_{12} + 217.5\,\lambda_{13} + 242.6\,\lambda_{14} + 304.0\,\lambda_{15}$

$+ 78.4\,\lambda_{22} + 244.0\,\lambda_{23} + 268.5\,\lambda_{24} + 321.2\,\lambda_{25} = 525$

$\lambda_{11} + \lambda_{12} + \lambda_{13} + \lambda_{14} + \lambda_{15} = 1$

$\lambda_{21} + \lambda_{22} + \lambda_{23} + \lambda_{24} + \lambda_{25} = 1$

Solution $\lambda_{11} = \lambda_{12} = \lambda_{13} = \lambda_{15} = 0,\ \lambda_{14} = 1.0$

$\lambda_{21} = \lambda_{22} = \lambda_{23} = 0,\ \lambda_{24} = 0.736,\ \lambda_{25} = 0.264$

Flow through generating unit 1 = 1.0 (242.6) = 242.6 m^3/s

Power produced by generating unit 1 = 1.0 (52.0) = 52 MW

Flow through generating unit 2 = 0.736 (268.5) + 0.264 (321.2) = 282.4 m^3/s

Power produced by generating unit 2 = 0.736 (57) + 0.264 (63) = 58.6 MW

Total power produced = 110.6 MW

APPENDIX 3

User Guide

This HYDOPT implementation is for the R.H. Saunders Hydraulic Generating Station. The station consists of 16 generating units of two different generating unit types. Units 1, 2, 4, 6, 8, 9, 10, 11, 12, 13, 14 and 15 are of one type and will be referred to as type one. Units 3, 5, 7, and 16 are of another type and will be referred to as type two. The HYDOPT system consists of the following files:

1. hydopt.exe - This is the executable file.

2. p1 - Each line in this file contains the 20 power points to be used for a range of heads, determined by the executable code, for type one units.

3. p2 - Each line in this file contains the 20 power points to be used for a range of heads, determined by the executable code, for type two units.

4. q1 - Each column indicates the flow, at a specific head determined by the executable file. For each line n in the file, the power associated with that flow is n-1 MW. This is for type one units. This file was provided by the Hydraulic Generation and Transmission Operations Division.

5. q2 - Each column indicates the flow, at a specific head determined by the executable file. For each line n in the file, the power associated with that flow is n-1 MW. This is for type two units. This file was provided by the Hydraulic Generation and Transmission Operations Division.

6. sepout - Contains the values of the grid point variables.

7. data - This file is listed below:

```
-----------------------------------------------------
16  2  20
1    1
2    1
3    0
4    1
5    2
6    1
7    2
9    1
10   1
11   1
12   1
13   1
14   1
15   1
16   2
2
-----------------------------------------------------
```

User Guide

The first three numbers on the first line represent the total number of generating units, the number of generating unit types and the number of points used to approximate each power curve. These numbers will not change. The next 16 lines represent the individual generating units. The first number represents the generating unit number and the second number represents the generating unit type. The number 1 represents generating unit type one. The number 2 represents generating unit type two. The number 0 tells HYDOPT that the generating unit is down for maintenance and no discharge is permitted to flow through it. The last number is the number of scaling iterations and need not be changed.

All these files must be in the same directory for HYDOPT to work. HYDOPT is interactive and will prompt the user to enter the head and flow. The user types in "hydopt" and then types in the head and flow when prompted. The following is an example of how to use HYDOPT.

hydopt <ENTER>

Enter the head in metres. (A decimal number)

25.18 <ENTER>

Enter the flow to be discharged in m^3/s. (A decimal number)

2654.3 <ENTER>

The head should be entered with two decimal places and the flow should be entered with one decimal place. HYDOPT will then output the discharge allocation and the power produced as shown below.

Unit 1	discharge	=	244.3 m^3/s	power	=	53.000 MW
Unit 2	discharge	=	244.3 m^3/s	power	=	53.000 MW
Unit 3	discharge	=	0.0 m^3/s	power	=	0.000 MW
Unit 4	discharge	=	244.3 m^3/s	power	=	53.000 MW
Unit 5	discharge	=	0.0 m^3/s	power	=	0.000 MW
Unit 6	discharge	=	244.2 m^3/s	power	=	52.979 MW
Unit 7	discharge	=	0.0 m^3/s	power	=	0.000 MW
Unit 8	discharge	=	239.6 m^3/s	power	=	52.000 MW
Unit 9	discharge	=	239.6 m^3/s	power	=	52.000 MW
Unit 10	discharge	=	239.6 m^3/s	power	=	52.000 MW
Unit 11	discharge	=	239.6 m^3/s	power	=	52.000 MW
Unit 12	discharge	=	0.0 m^3/s	power	=	0.000 MW
Unit 13	discharge	=	239.6 m^3/s	power	=	52.000 MW
Unit 14	discharge	=	239.6 m^3/s	power	=	52.000 MW
Unit 15	discharge	=	239.6 m^3/s	power	=	52.000 MW
Unit 16	discharge	=	0.0 m^3/s	power	=	0.000 MW
Total discharge		=	2654.3 m^3/s	Total power	=	575.979 MW

APPENDIX 4

Function Definitions

colyk	-	Calculates the tableau column of the entering nonbasis variable at the current iteration.
varleave	-	Determines the basic variable to leave the basis during the separable programming portion of the algorithm.
lpvarleave	-	Determines the basic variable to leave the basis during the linear programming portion of the algorithm.
wzjcj	-	Determines the nonbasic variable to enter the basis during the separable programming portion of the algorithm.
lpwzjcj	-	Determines the nonbasic variable to enter the basis during the linear programming portion of the algorithm.
scale	-	Scales the objective function and constraints using a geometric mean scaling approach. This reduces numerical instability.
rescale	-	Rescales the solution.
generate	-	Generates the objective function, constraint matrix and the right hand side.
readflow	-	Reads the power points and flow points from the p1, p2, q1 and q2 files for a user specified head.
calcflow	-	Interpolates the flows for specific power points for a user specified head.
round1st	-	Rounds mathematically to one decimal place.
qflow	-	Calculates the discharge through each generating unit based on the separable or linear programming results.
active	-	Determines the highest slope segment that has water for each generating unit.
slopealloc	-	Reallocates water from a shutdown generating unit to other generating units currently discharging water. This is based on the slope segments of the power curves.
mwpower	-	Calculates the power produced by the current discharge allocation.
finetune	-	Attempts to increase power by reallocating water among generating units that are currently discharging water. This is based on the slope segments of the power curves.

APPENDIX 4 (Cont'd)

Function Definitions

postsep - Calls the function finetune after the separable programming portion of the algorithm is done.

postlp1 - The generating unit with the nonadjacent points is shutdown. The water is reallocated using the function slopealloc. This occurs after the linear programming portion of the algorithm is complete.

postlp2 - The discharge allocation from the linear programming solution is reallocated using the function finetune.

APPENDIX 5

Flow Diagram

The following flow diagram was produced using the UNIX utility cflow.

```
main:
    readflow:
        calcflow:
    generate:
    scale:
    colyk:
    varleave:
    lpvarleave:
    wzjcj:
    lpwzjcj:
    postsep:
        qflow:
        active:
        finetune:
        wpower:
        slopealloc:
    rescale:
    postlp1:
        qflow:
        active:
        mwpower:
        slopealloc:
        finetune:
    postlp2:
        qflow:
        active:
        finetune:
        mwpower:
```

APPENDIX 6

Heuristics

Function slopealloc

1. Shutdown specified unit.
2. Locate available slope segment with maximum gradient.
3. Allocate water from shutdown unit to slope segment with maximum gradient.
4. If any water from shutdown unit remains, go to 2.

Function finetune

1. Locate available slope segment with maximum gradient.
2. Locate occupied slope segment with minimum gradient.
3. If maximum gradient > minimum gradient.
4. Begin
5. Allocate water from slope segment with minimum gradient to slope segment with maximum gradient.
6. While water still remains from slope segment with minimum gradient.
7. Begin.
8. Locate available slope segment with maximum gradient.
9. Allocate water to slope segment with maximum gradient
10. End.
11. End
12. Locate available slope segment with maximum gradient.
13. Locate occupied slope segment with minimum gradient.
14. If maximum gradient > minimum gradient go to 4.

APPENDIX 7

Variable Definitions

m	- The number of constraints.
n	- The number of variables.
naug	- The augmented number of variables.
i, j, k, kl, r	- Loop index variables.
iter	- Contains the current iteration count.
nonbasic	- The number of nonbasic variables.
optimal	- Equals 1 if optimal or 0 if not optimal.
leave	- Contains the variable to leave the basis.
enter	- Contains the variable to enter the basis.
unbounded	- Equals 1 if unbounded or 0 if bounded.
temp	- Used for temporary storage of variables.
eindex[i]	- Contains the column number of the elementary matrix used in ith iteration of the product form of the inverse.
xbindex[i]	- Each location in the vector contains one of the basic variable indexes.
xnindex[i]	- Each location in the vector contains one of the nonbasic variable indexes.
phase1	- Equals 1 if the algorithm is in phase 1 or 0 if it is in phase 2.
artificials	- The number of artificial variables.
infeasible	- Equals 1 if the problem is infeasible or 0 if it is feasible.
p1iter	- The number of phase 1 iterations.
mle	- The number of less than or equal to constraints.
mge	- The number of greater than or equal to constraints.
meq	- The number of equality constraints.
counter	- The number of tied variables that can possibly leave the basis.
tievar[i]	- Contains the indexes of the tied variables that can possibly leave the basis.
randvar[i]	- Contains the random numbers used to break the tie if more than one variable can leave the basis.
iterscale	- The number of scaling iterations.
unittype[i]	- Contains the generating unit type of generating unit i.
unitinfo[i][0]	- Contains the number of points that approximate the power curve of generating unit i. (A maximum of 2 and they must be adjacent)
unitinfo[i][1].	- Contains the first variable to approximate the power curve of generating unit i.
unitinfo[i][2]	- Contains the second variable to approximate the power curve of generating unit i.
unit	- Specifies a particular generating unit number.
numop	- The number of generating units available to discharge water.
numtypes	- The number of generating unit types.
numunits	- The total number of generating units in the station.
umpoints	- The total number of points used for all the power curves.

Variable Definitions

shutdown	- The generating unit shutdown in an attempt to produce more power.
segment[i]	- Contains the number of the highest slope segment with water in it for generating unit i.
sepprog	- Equals 1 if the algorithm is in the separable programming portion or 0 if it is in the linear programming portion.
c[i]	- Contains the objective function coefficient for variable i.
b[i]	- Contains the right hand side value for constraint i.
a[i][j]	- Contains the value of the i and j element of the constraint matrix.
xn	- Contains the value of the nonbasic variables, zero.
co[i]	- Used to store the original objective function coefficient for variable i.
xpointer[i]	- Contains the address of the value of variable i.
zcmin	- The minimum reduced cost when selecting an entering variable into the basis.
by[i]	- The column vector of the entering nonbasic variable in the current tableau.
colmin	- The minimum ratio when selecting a basic variable to leave the basis.
e[i][j]	- Contains the j column vectors needed in the product form of the inverse representation.
w[i]	- Intermediate computation vector to calculate the reduced costs in the product form of the inverse.
sum	- Variable used to sum various quantities.
zc	- Contains the value of the current reduced cost.
y[i]	- Contains the vector of the current tableau whose variable is entering the basis.
ctx	- Contains the value of the objective function at the current iteration.
btemp	- Used to contain temporary values.
ytemp	- Used to contain temporary values.
colscale[i]	- The value column i was scaled by.
rowscale[i]	- The value row i was scaled by.
power[i][j]	- The power value for unit type i at point j.
flow[i][j]	- The flow value for unit type i at point j.
qtotal	- The total water to be discharged.
head	- The head.
flowc[i][j]	- The flow curves for a specific head from the files q1 and q2. Flows for a specific head j that produces i-1 MW. These curves are used for interpolation purposes.
q[i]	- The water discharged through generating unit i.
q[0]	- The total water discharged through the generating station.
mw[i]	- The power produced from generating unit i.
mw[0]	- The total power produced by the generating station.

APPENDIX 7 (Cont'd)

Variable Definitions

qbest[i]	- The discharge allocation that produces the most power for the i generating units to this point in the algorithm.
qbest[0]	- The total water discharged through the generating station.
mwbest[i]	- The most power produced by the i generating units to this point in the algorithm.
mwbest[0]	- The total power produced by the generating station.
slope[i][j]	- The slope of the power curve of segment j of generating unit type i.
unitsup[i]	- Stores the unittype[i] vector when units are being shutdown.
down	- Contains the unit that is being shutdown.
minunit	- Contains the unit with the minimum slope where water has been allocated.
qremain	- The amount of water which has yet to be allocated.
minslope	- The minimum slope where water has been allocated among the active units.
flowdown	- The amount of water to be reallocated.
rowmax	- The maximum element in a row of the constraint matrix.
rowmin	- The minimum element in a row of the constraint matrix.
colmax	- The maximum element in a column of the constraint matrix.
typeunit	- Contains the type of a particular unit.
lineptr	- A line counter for the input files.
headc[i]	- Contains the head values for measured power and flow points.
dummy[260]	- Used to store records that are skipped in input files.
dummy	- A dummy pointer used in skipping records in input files.
equal	- Specifies whether the head equals one of the measured heads.
isoline	- Contains the isoline number used.
iflow	- Integer value of the flow. (Rounded)
ipower	- Integer value of the power. (Rounded)
flowhi	- The flow value for the isoline above the given head.
flowlo	- The flow value for the isoline below the given head.
percent	- The percent which a head is between two isolines.
fraction	- The fraction which a power point is between two power points.
xr	- The value of the flow to be rounded to one decimal place.
xi	- The truncated integer value of the flow.
yr	- The truncated float value of the flow.
kunit	- Contains the unit number to which water is allocated.
allocate	- The amount of water remaining to be allocated.

APPENDIX 7 (Cont'd)

Variable Definitions

qq	- Temporary variable containing the flow allocated to that point.
maxslope	- The maximum slope of the units which is as yet unallocated.
maxunit	- The unit containing the maximum unallocated slope.
available	- The amount of water available for reallocation.
fpr	- File pointer to the "data" file.
fp	- File pointer to the "sepout" file.
pp1	- File pointer to the "p1" file.
pp2	- File pointer to the "p2" file.
fp1	- File pointer to the "q1" file.
fp2	- File pointer to the "q2" file.

APPENDIX 8

HYDOPT Listing

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MMAX 34
#define NMAX 370
#define EPS 1.0e-8
#define MAXTYPES  3
#define MAXPOINTS 21
#define MAXUNITS 17

FILE *fp, *fpr;

int m, n, naug, i, j, iter, nonbasic, optimal,
   leave, enter, unbounded, temp, eindex[NMAX],
   xbindex[NMAX], xnindex[NMAX], k, phase1,
   artificials, infeasible, p1iter, mle, mge,
   meq, counter, tievar[MMAX], randvar[MMAX],
   iterscale, unittype[MAXUNITS],
   unitinfo[MAXUNITS][4], unit, numunits, numtypes,
   numop, numpoints, shutdown, segment[MAXUNITS], sepprog;

double c[NMAX], b[MMAX], a[MMAX][NMAX], xn, co[NMAX],
     *xpointer[NMAX], zcmin, by[MMAX], colmin, e[MMAX][NMAX],
     w[NMAX], sum, zc, y[MMAX], ctx, btemp, ytemp,
     colscale[NMAX], rowscale[MMAX], power[MAXTYPES][MAXPOINTS],
     flow[MAXTYPES][MAXPOINTS], qtotal, head, flowc[73][22],
     q[MAXUNITS], mw[MAXUNITS], qbest[MAXUNITS], mwbest[MAXUNITS],
     slope[MAXTYPES][MAXPOINTS];

void colyk(void);
void varleave(void);
void lpvarleave(void);
void wzjcj(void);
void lpwzjcj(void);
void scale(void);
void rescale(void);
void generate(void);
void readflow(void);
void calcflow(int typeunit, double headc[22]);
double round1st(double xr);
void qflow(void);
```

APPENDIX 8

HYDOPT Listing

```
void active(void);
void slopealloc(void);
void mwpower(void);
void postsep(void);
void postlp1(void);
void postlp2(void);
void finetune(void);



void main(void)
{
/* Open input file */
fpr = fopen("data","r");

/* Open output file */
fp = fopen("sepout","w");

/* Read in characteristics of generating station */
fscanf(fpr,"%d %d %d",&numunits,&numtypes,&numpoints);

for(i=1; i<=numunits; ++i)
  {
   fscanf(fpr,"%d %d",&temp,&unittype[i]);
  }

/* Read in # of iteration scales */
     fscanf(fpr,"%d",&iterscale);

fclose(fpr);

/* Calculate the number of units available */
numop = 0;
for(i=1; i<=numunits; ++i)
{
   if( unittype[i] > 0 ) ++numop;
}

/* Calculate LP dimensions */
mle = numop + 1;
mge = 0;
meq = numop;
n = numop * numpoints;

m = mle + mge + meq;
naug = n + mle + 2 * mge + meq;
nonbasic = naug - m;
artificials = mge + meq;
```

```
printf("Enter the head in metres. (A decimal number)\n");
scanf("%lf",&head);

printf("Enter the flow to be discharged in m3/s. (A decimal number)\n");
scanf("%lf",&qtotal);

/* Calculate the flows of the objective function */
readflow();


/* Generate constraints, objective function and RHS */
generate();


/* Check for negative right hand side */
for(i=1; i<=m; ++i)
  if (b[i] < 0.0 )
    {
     printf("The right hand side has negative components.\n");
     printf("Reformulate the problem with non-negative RHS.\n");
     exit(0);
    }


/* Scale problem */
if ( iterscale > 0 ) scale();

/* Zero out unit information matrix */
for(i=1; i<=numop; ++i)
  for(j=0; j<=numtypes; ++j)
    unitinfo[i][j] = 0;


iter = 0;
optimal = 0;
xn = 0.0;
sepprog = 1;


/* Zero out augmented matrix */
for(i=1; i<=m; ++i)
  for(j=n+1; j<=naug; ++j)
    a[i][j] = 0.0;


/* Zero out additional variables in original objective function */
k = naug - artificials;
```

```
for(j=n+1; j<=k; ++j)
  co[j] = 0.0;


/* Set slack variables & identify basic variable positions */
j = n + 1;
for(i=1; i<=mle; ++i)
 {
  a[i][j] = 1.0;
  xbindex[i] = j;
  xpointer[j] = &b[i];
  ++j;
 }


if (artificials > 0)
 {
  phase1 = 1;

/* Set surplus variables */
  j = n + mle + 1;
  k = mle + mge;
  for(i=mle+1; i<=k; ++i)
   {
    a[i][j] = -1.0;
    ++j;
   }


/* Set artificial variables & identify basic variable positions */
  j = n + mle + mge + 1;
  (mle > 0) ? (k = mle + 1) : (k = 1);
  for(i=mle+1; i<=m; ++i)
   {
    a[i][j] = 1.0;
    xbindex[k] = j;
    xpointer[j] = &b[i];
    ++j;
    ++k;
   }


/* Calculate Phase I objective function */
  k = n + mle + mge;
  for(j=1; j<=k; ++j)
   {
    sum = 0.0;
    for(i=mle+1; i<=m; ++i)
```

```
        sum = sum + a[i][j];
      c[j] = sum;
    }


/* Zero out remaining objective function variables */
   for(j=n+mle+mge+1; j<=naug; ++j)
      c[j] = 0.0;
  }


else
  {
   phase1 = 0;
   p1iter = 0;

/* Restore objective function to original form */
   for(j=1; j<=naug; ++j)
      c[j] = co[j];
  }


/* Set nonbasic variables and identify position */
for(i=1; i<=n; ++i)
  {
   xnindex[i] = i;
   xpointer[i] = &xn;
  }
if (mge > 0)
  {
   j = n + 1;
   k = n + mle + 1;
   for(i=1; i<=mge; ++i)
     {
      xnindex[j] = k;
      xpointer[k] = &xn;
      ++j;
      ++k;
     }
  }


/* Determine entering variable into basis */
enter = 0;
zcmin = 0.0;
for(j=1; j<=nonbasic; ++j)
  {
   if ( c[xnindex[j]] > 0.0 && c[xnindex[j]] > zcmin )
```

```
    {
     if( xnindex[j] <= n )
       {
        if( (xnindex[j] % numpoints) > 0 )
          unit = xnindex[j]/numpoints + 1;
        else
          unit = xnindex[j]/numpoints;

        if ( unitinfo[unit][0] == 0 )
          {
           zcmin = c[xnindex[j]];
           enter = j;
          }
        else if ( unitinfo[unit][0] == 1 )
          {
           temp = abs( unitinfo[unit][1] - xnindex[j] );
           if ( temp == 1 )
             {
             zcmin = c[xnindex[j]];
             enter = j;
             }
          }
       }
      else
       {
        zcmin = c[xnindex[j]];
        enter = j;
       }
     }
  }
if( xnindex[enter] <= n )
  {
   if( (xnindex[enter] % numpoints) > 0 )
     unit = xnindex[enter]/numpoints + 1;
   else
     unit = xnindex[enter]/numpoints;

   if ( unitinfo[unit][0] == 0 )
     {
      unitinfo[unit][1] = xnindex[enter];
      ++unitinfo[unit][0];
     }
   else
     {
      unitinfo[unit][2] = xnindex[enter];
      ++unitinfo[unit][0];
     }
  }
```

```
if (enter == 0)
 {
  printf("Optimal objective function value is 0.\n");
  printf("Check problem formulation.\n");
  exit(0);
 }

/********************* Beginning of main loop ***************************/
while (optimal == 0)
 {
  ++iter;

  if (iter == NMAX)
   {
    printf("Maximum number of iterations has been exceeded.\n");
    printf("The array size of E must be changed in the source code.\n");
    exit(0);
   }


  colyk();

  if( sepprog == 1 )
   {
    varleave();
   }
  else
   {
    lpvarleave();
   }

  printf("Iter = %3d  leaving variable x(%3d)  counter = %d\n",
       iter,xbindex[leave],counter);


  if (unbounded == 1)
   {
    printf("The problem is unbounded.\n");
    exit(0);
   }


/* Calculate nonidentity column of E */
  for (i=1; i<=m; ++i)
    if (i != leave)
```

```
        e[i][iter] = -y[i]/y[leave];
      else
        e[i][iter] = 1.0/y[leave];
    eindex[iter] = leave;


/* Update right hand side */
    btemp = b[leave];
    for(i=1; i<=m; ++i)
      if (i != leave)
        b[i] = b[i] + e[i][iter] * btemp;
      else
        b[i] = e[i][iter] * btemp;


/* Check right hand side for negative values and zero them out */
    for(i=1; i<=m; ++i)
      if ( b[i] < 0.0 )
        b[i] = 0.0;


/* Update basic and nonbasic variable storage locations */
    temp = xbindex[leave];
    xbindex[leave] = xnindex[enter];
    xnindex[enter] = temp;
    xpointer[xbindex[leave]] = &b[leave];
    xpointer[xnindex[enter]] = &xn;


/* Calculate objective function value */
    ctx = 0.0;
    if ( phase1 == 0 )
      {
       for(i=1; i<=m; ++i)
          ctx = ctx + c[xbindex[i]] * b[i];
       printf("   Objective Function Value = %17.8lf\n", ctx);
      }
    else
      {
       for(j=naug-artificials+1; j<naug; ++j)
          ctx += *xpointer[j];
       printf("   Objective Function Value = %17.8lf\n", ctx);
      }


    if( sepprog == 1 )
      {
       wzjcj();
```

```
  }
 else
  {
   lpwzjcj();
  }

 if (enter == 0)
  {
   if (phase1 == 1)
    {
     p1iter = iter;


/* Check for infeasibility */
      infeasible = 0;
      for(j=naug-artificials+1; j<=naug; ++j)
        if ( *xpointer[j] >= EPS )
          infeasible = 1;


      if (infeasible == 1)
       {
        printf("The problem is infeasible.\n");
        exit(0);
       }


/* Check for artifical variables in the basis */
      k = naug - artificials;
      for(i=1; i<=m; ++i)
        {
         if( xbindex[i] >  k)
          {
           printf("An artificial variable is in the basis.\n");
           printf("Equation %3d\n.",i);
           exit(0);
          }
        }


/* Eliminate artificial variables from nonbasic index */
      i = naug - artificials;
      k = 1;
      for(j=1; j<=nonbasic; ++j)
        if ( xnindex[j] <= i )
          {
           xnindex[k] = xnindex[j];
           ++k;
```

```
        }

/* Eliminate artificial variable dimensions of LP */
        naug -= artificials;
        nonbasic -= artificials;
        phase1 = 0;
        printf("Phase II\n");


/* Restore objective function to original form */
        for(j=1; j<=naug; ++j)
          c[j] = co[j];


        wzjcj();


/* Check for optimality after Phase I */
        if (enter == 0)
          optimal = 1;

       }
      else
       {
        if( sepprog == 1 )
         {
          postsep();
          lpwzjcj();
          sepprog = 0;
          if( enter == 0 ) optimal = 1;
         }
        else
         {
          optimal = 1;
         }
       }
     }
  }
/****************** End of main loop ***********************************/

/* Calculate objective function value */
ctx = 0.0;
for(i=1; i<=m; ++i)
   ctx = ctx + c[xbindex[i]] * b[i];


if( iterscale > 0 ) rescale();
```

```
/* Print out final solution */
fprintf(fp,"Phase I  Iterations = %3d\n", p1iter);
fprintf(fp,"Phase II Iterations = %3d\n", iter - p1iter);
fprintf(fp,"Objective Function Value = %17.8lf\n", ctx);
for(j=1; j<=n; ++j)
  fprintf(fp,"x(%3d) = %17.8lf\n", j, *xpointer[j]);

/* Close output file */
fclose(fp);

postlp1();

if( shutdown > 0 ) postlp2();

for(i=1; i<=numunits; ++i)
  printf("Unit %2d   discharge = %7.1lf m3/s   power = %8.3lf MW \n",
       i, qbest[i], mwbest[i] );

printf("Total discharge = %7.1lf m3/s   Total power = %8.3lf MW\n",
     qbest[0], mwbest[0]);

printf("Shutdown = %3d\n",shutdown);

}



void lpvarleave(void)
{
/* Determine leaving variable from the basis */
k = 1;
unbounded = 1;
for(i=1; i<=m; ++i)
  if (y[i] >= EPS)
    {
     by[i] = b[i]/y[i];
     if ( k > 1 )
       {
        if ( by[i] < colmin )
          {
           colmin = by[i];
           leave = i;
          }
       }
     else
       {
```

```
        colmin = by[i];
        unbounded = 0;
        leave = i;
        ++k;
       }
    }
counter = 1;
if ( unbounded == 0 )
  for(i=1; i<=m; ++i)
    if (y[i] >= EPS)
      if( by[i] == colmin  && (i != leave) )
        {
         if ( counter == 1 )
          {
           tievar[1] = leave;
           tievar[2] = i;
           randvar[1] = rand();
           randvar[2] = rand();
           ++counter;
          }
         else
          {
           ++counter;
           tievar[counter] = i;
           randvar[counter] = rand();
          }
        }
 if ( counter > 1 )
  {
   j = 1;
   for(i=2;i<=counter;++i)
    {
     if( randvar[i] < randvar[j] )
      {
       leave = tievar[i];
       j = i;
      }
    }
   for(i=1;i<=counter;++i)
    {
     printf("var = %d     randvar = %d\n",xbindex[tievar[i]],randvar[i]);
    }
   printf("leave = %d\n",xbindex[leave]);
  }
 }
```

```
void lpwzjcj(void)
{
/* Update w */
for(i=1; i<=m; ++i)
  if (i != eindex[iter])
    w[i] = c[xbindex[i]];
  else
   {
    sum = 0.0;
    for(j=1; j<=m; ++j)
      sum = sum + c[xbindex[j]] * e[j][iter];
    w[i] = sum;
   }
if (iter > 1)
  for(k=iter-1; k!=0; --k)
    for(i=1; i<=m; ++i)
      if (i == eindex[k])
       {
        sum = 0.0;
        for(j=1; j<=m; ++j)
          sum = sum + w[j] * e[j][k];
        w[i] = sum;
       }


/* Compute Zj - Cj's and determine entering basis variable */
enter = 0;
zcmin = 0.0;
for(j=1; j<=nonbasic; ++j)
 {
  zc = 0.0;
  for(i=1; i<=m; ++i)
    zc = zc + w[i] * a[i][xnindex[j]];
  zc = zc - c[xnindex[j]];
  if (zc <= -EPS && zc < zcmin)
   {
    zcmin = zc;
    enter = j;
   }
 }
}



void postsep(void)
{
int unitsup[MAXUNITS];
```

```
/* Store units available */
for(i=1; i<=numunits; ++i)
 {
  unitsup[i] = unittype[i];
 }

/* Rescale the solution */
for(i=1; i<=m; ++i)
 {
  b[i] /= colscale[xbindex[i]];
 }

qflow();
active();

/* Calculate slopes of power curves */
for(i=1; i<=numtypes; ++i)
 {
  for(j=1; j<numpoints; ++j)
   {
    slope[i][j] = ( power[i][j+1] - power[i][j] ) /
                  ( flow[i][j+1] - flow[i][j]   );
   }
 }

finetune();
mwpower();

for(i=0; i<=numunits; ++i)
 {
  qbest[i] = q[i];
  mwbest[i] = mw[i];
 }

/* Determine units to shutdown */
shutdown = 0;
for(i=1; i<=numunits; ++i)
 {
  if(  unittype[i] > 0 && q[i] < flow[unittype[i]][3]  )
   {
    q[i] = 0.0;
    unittype[i] = 0;
    segment[i] = 1;
    shutdown = i;
   }
 }

if( shutdown > 0 )
```

```
 {
  slopealloc();
  finetune();
  mwpower();
  if( mw[0] > mwbest[0] )
    {
     for(i=0; i<=numunits; ++i)
       {
        qbest[i] = q[i];
        mwbest[i] = mw[i];
       }
    }
 }

/* Restore units available */
for(i=1; i<=numunits; ++i)
 {
  unittype[i] = unitsup[i];
 }


/* Rescale the solution */
for(i=1; i<=m; ++i)
 {
  b[i] *= colscale[xbindex[i]];
 }
}


void postlp1(void)
{
int down;

qflow();
active();
mwpower();

if( mw[0] > mwbest[0] )
 {
  for(i=0; i<=numunits; ++i)
    {
     qbest[i] = q[i];
     mwbest[i] = mw[i];
    }
 }

down = 0;
for(i=1; i<=n; ++i)
```

```
  {
   if( 0.00000001 <= *xpointer[i] && *xpointer[i] <= 0.99999999 )
     {
      down = i;
      break;
     }
  }

if( down == 0 )
  {
   shutdown = 0;
   return;
  }

if( (down % numpoints) > 0 )
   shutdown = i/numpoints + 1;
else
   shutdown = i/numpoints;

i = 1;
k = 0;
while( k != shutdown )
  {
   if( unittype[i] != 0 ) ++k;
   if( k != shutdown ) ++i;
  }
shutdown = i;

q[shutdown] = 0.0;
unittype[0] = unittype[shutdown];
unittype[shutdown] = 0;
segment[shutdown] = 1;

slopealloc();
finetune();
mwpower();

if( mw[0] > mwbest[0] )
  {
   for(i=0; i<=numunits; ++i)
     {
      qbest[i] = q[i];
      mwbest[i] = mw[i];
     }
  }

unittype[shutdown] = unittype[0];
}
```

```
void postlp2(void)
{
int minunit;

double qremain, minslope, flowdown;

qflow();
active();

/* 0.1 added to prevent finetune() from shutting unit down */
if( q[shutdown] < (flow[unittype[shutdown]][2]+0.1) )
 {
  qremain = flow[unittype[shutdown]][2] - q[shutdown] + 0.1;
  while( qremain > 0.0 )
   {
    minslope = 1.0e8;
    for(i=1; i<=numunits; ++i)
     {
      if( unittype[i] > 0 && q[i] > 0.0 && i != shutdown )
       {
        if( q[i] == flow[unittype[i]][segment[i]] )
         {
          if( slope[unittype[i]][segment[i]-1] < minslope )
           {
            minslope = slope[unittype[i]][segment[i]-1];
            minunit = i;
           }
         }
        else
         {
          if( slope[unittype[i]][segment[i]] < minslope )
           {
            minslope = slope[unittype[i]][segment[i]];
            minunit = i;
           }
         }
       }
     }

    if( q[minunit] == flow[unittype[minunit]][segment[minunit]] )
      flowdown = q[minunit] -
              flow[unittype[minunit]][segment[minunit]-1];
    else
      flowdown = q[minunit] -
              flow[unittype[minunit]][segment[minunit]];
```

```
    if( flowdown < qremain )
      {
       if( q[minunit] == flow[unittype[minunit]][segment[minunit]] )
          --segment[minunit];
       q[shutdown] += flowdown;
       q[minunit] -= flowdown;
       qremain -= flowdown;
      }
    else
      {
       if( q[minunit] == flow[unittype[minunit]][segment[minunit]] )
          --segment[minunit];
       q[shutdown] += qremain;
       q[minunit] -= qremain;
       qremain = 0.0;
       ++segment[shutdown];
      }
   }
 }

finetune();
mwpower();

if( mw[0] > mwbest[0] )
 {
  for(i=0; i<=numunits; ++i)
   {
    qbest[i] = q[i];
    mwbest[i] = mw[i];
   }
 }

}



void colyk(void)
{
/* Calculate column Yk */
if (iter != 1)
 {
  for(i=1; i<=m; ++i)
    if (i != eindex[1])
      y[i] = a[i][xnindex[enter]]
           + a[eindex[1]][xnindex[enter]] * e[i][1];
    else
      y[i] = a[eindex[1]][xnindex[enter]] * e[i][1];
```

```
  if (iter > 2)
    for(j=2; j<iter; ++j)
     {
      ytemp = y[eindex[j]];
      for(i=1; i<=m; ++i)
        if (i != eindex[j])
          y[i] = y[i] + ytemp * e[i][j];
        else
          y[i] = ytemp * e[i][j];
     }
 }
else
  for(i=1; i<=m; ++i)
    y[i] = a[i][enter];
}



void varleave(void)
{
/* Determine leaving variable from the basis */
k = 1;
unbounded = 1;
for(i=1; i<=m; ++i)
  if (y[i] >= EPS)
   {
    by[i] = b[i]/y[i];
    if ( k > 1 )
     {
      if ( by[i] < colmin )
       {
        colmin = by[i];
        leave = i;
       }
     }
    else
     {
      colmin = by[i];
      unbounded = 0;
      leave = i;
      ++k;
     }
   }
counter = 1;
if ( unbounded == 0 )
  for(i=1; i<=m; ++i)
    if (y[i] >= EPS)
      if( by[i] == colmin  && (i != leave) )
```

```
        {
         if ( counter == 1 )
           {
            tievar[1] = leave;
            tievar[2] = i;
            randvar[1] = rand();
            randvar[2] = rand();
            ++counter;
           }
         else
           {
            ++counter;
            tievar[counter] = i;
            randvar[counter] = rand();
           }
        }
if ( counter > 1 )
  {
   j = 1;
   for(i=2;i<=counter;++i)
     {
      if( randvar[i] < randvar[j] )
        {
         leave = tievar[i];
         j = i;
        }
     }
   for(i=1;i<=counter;++i)
     {
      printf("var = %d     randvar = %d\n",xbindex[tievar[i]],randvar[i]);
     }
   printf("leave = %d\n",xbindex[leave]);
  }
if ( xbindex[leave] <= n )
  {
   if( (xbindex[leave] % numpoints) > 0 )
      unit = xbindex[leave]/numpoints + 1;
   else
      unit = xbindex[leave]/numpoints;

   if ( unitinfo[unit][0] == 1 )
     {
      unitinfo[unit][1] = 0;
     }
   else
     {
      if ( unitinfo[unit][1] == xbindex[leave] )
        {
```

```
        unitinfo[unit][1] = unitinfo[unit][2];
        unitinfo[unit][2] = 0;
       }
      else
       {
        unitinfo[unit][2] = 0;
       }
     }
   --unitinfo[unit][0];
  }
}


void wzjcj(void)
{
/* Update w */
for(i=1; i<=m; ++i)
  if (i != eindex[iter])
    w[i] = c[xbindex[i]];
  else
   {
    sum = 0.0;
    for(j=1; j<=m; ++j)
      sum = sum + c[xbindex[j]] * e[j][iter];
    w[i] = sum;
   }
if (iter > 1)
  for(k=iter-1; k!=0; --k)
    for(i=1; i<=m; ++i)
      if (i == eindex[k])
       {
        sum = 0.0;
        for(j=1; j<=m; ++j)
          sum = sum + w[j] * e[j][k];
        w[i] = sum;
       }


/* Compute Zj - Cj's and determine entering basis variable */
enter = 0;
zcmin = 0.0;
for(j=1; j<=nonbasic; ++j)
 {
  zc = 0.0;
  for(i=1; i<=m; ++i)
    zc = zc + w[i] * a[i][xnindex[j]];
  zc = zc - c[xnindex[j]];
```

```
 if (zc <= -EPS && zc < zcmin)
  {
   if( xnindex[j] <= n )
    {
     if( (xnindex[j] % numpoints) > 0 )
       unit = xnindex[j]/numpoints + 1;
     else
       unit = xnindex[j]/numpoints;

     if ( unitinfo[unit][0] == 0 )
       {
        zcmin = zc;
        enter = j;
       }
     else if ( unitinfo[unit][0] == 1 )
       {
        temp = abs( unitinfo[unit][1] - xnindex[j] );
        if ( temp == 1 )
          {
           zcmin = zc;
           enter = j;
          }
       }
    }
   else
    {
     zcmin = zc;
     enter = j;
    }
  }
 }
if( xnindex[enter] <= n )
 {
  if( (xnindex[enter] % numpoints) > 0 )
    unit = xnindex[enter]/numpoints + 1;
  else
    unit = xnindex[enter]/numpoints;

  if ( unitinfo[unit][0] == 0 )
    {
     unitinfo[unit][1] = xnindex[enter];
     ++unitinfo[unit][0];
    }
  else
    {
     unitinfo[unit][2] = xnindex[enter];
     ++unitinfo[unit][0];
    }
```

```
  }
}


void scale(void)
{
int r, k1;

double rowmax, rowmin, colmax;

for(i=1; i<=m; ++i)
   rowscale[i] = 1.0;

for(j=1; j<=naug; ++j)
   colscale[j] = 1.0;

for(k1=1; k1<=iterscale; ++k1)
  {
/* Perform row scaling  */
   for(i=1; i<=m; ++i)
     {
      k = 0;
      for(j=1; j<=n; ++j)
        {
         if ( k > 0 && a[i][j] != 0.0 )
           {
            if ( fabs(a[i][j]) > rowmax ) rowmax = fabs(a[i][j]);
            else if ( fabs(a[i][j]) < rowmin ) rowmin = fabs(a[i][j]);
           }
         if ( k == 0 )
           {
            if ( a[i][j] != 0.0 )
              {
                rowmax = fabs(a[i][j]);
                rowmin = fabs(a[i][j]);
               ++k;
              }
           }
        }
      rowscale[0] = sqrt( rowmax * rowmin );
      rowscale[i] =  rowscale[i] * rowscale[0];

      for(r=1; r<=n; ++r)
         a[i][r] = a[i][r] / rowscale[0];
      b[i] = b[i] / rowscale[0];

     }
```

```
 /* Perform column scaling */
 for(j=1; j<=n; ++j)
  {
   k = 0;
   for(i=1; i<=m; ++i)
    {
     if ( k > 0 && a[i][j] != 0.0 )
      {
       if ( fabs(a[i][j]) > colmax ) colmax = fabs(a[i][j]);
       else if ( fabs(a[i][j]) < colmin ) colmin = fabs(a[i][j]);
      }
     if ( k == 0 )
      {
       if ( a[i][j] != 0.0 )
        {
          colmax = fabs(a[i][j]);
          colmin = fabs(a[i][j]);
         ++k;
        }
      }
    }
   colscale[0] = sqrt( colmax * colmin );
   colscale[j] = colscale[j] * colscale[0];

   for(r=1; r<=m; ++r)
     a[r][j] = a[r][j] / colscale[0];
   co[j] = co[j] / colscale[0];
  }
 }

}



void rescale(void)
{
for(i=1; i<=m; ++i)
  b[i] = b[i] / colscale[xbindex[i]];
}



void generate(void)
{
int r;

for(i=1; i<=numop; ++i)
```

```
   for(j=1; j<=n; ++j)
     a[i][j] = 0.0;

j = 1;
r = 1;
for(i=1; i<=numop; ++i)
  {
   while( unittype[r] == 0 )
     {
      ++r;
     }
   for(k=1; k<=numpoints; ++k)
     {
      a[i][j] = flow[unittype[r]][k];
      co[j] = power[unittype[r]][k];
      ++j;
     }
   b[i] = flow[unittype[r]][numpoints];
   ++r;
  }

j = 1;
r = 1;
for(i=1; i<=numop; ++i)
  {
   while( unittype[r] == 0 )
     {
      ++r;
     }
   for(k=1; k<=numpoints; ++k)
     {
      a[numop+1][j] = flow[unittype[r]][k];
      ++j;
     }
   ++r;
  }
b[numop+1] = qtotal;

for(i=mle+1; i<=m; ++i)
   for(j=1; j<=n; ++j)
     a[i][j] = 0.0;

j = 1;
for(i=mle+1; i<=m; ++i)
  {
   for(k=1; k<=numpoints; ++k)
     {
      a[i][j] = 1.0;
```

```
    ++j;
   }
  b[i] = 1.0;
 }

}


void readflow(void)
{
int typeunit, lineptr;

double headc[22];

char dummy[260], *dummy2;

FILE *fp1, *fp2, *pp1, *pp2;

/* Heads used to measure specific powers and flows */
headc[1]  = 21.3360;
headc[2]  = 21.6408;
headc[3]  = 21.9456;
headc[4]  = 22.2504;
headc[5]  = 22.5552;
headc[6]  = 22.8600;
headc[7]  = 23.1648;
headc[8]  = 23.4696;
headc[9]  = 23.7744;
headc[10] = 24.0792;
headc[11] = 24.3840;
headc[12] = 24.6888;
headc[13] = 24.9936;
headc[14] = 25.2984;
headc[15] = 25.6032;
headc[16] = 25.9080;
headc[17] = 26.2128;
headc[18] = 26.5176;
headc[19] = 26.8224;
headc[20] = 27.1272;
headc[21] = 27.4320;


/* Determine line of power points to read in */
if      ( 21.40 <= head && head < 21.70 )
   lineptr = 0;
else if ( 21.70 <= head && head < 22.00 )
   lineptr = 1;
```

```
else if ( 22.00 <= head && head < 22.30 )
  lineptr = 2;
else if ( 22.30 <= head && head < 22.60 )
  lineptr = 3;
else if ( 22.60 <= head && head < 22.90 )
  lineptr = 4;
else if ( 22.90 <= head && head < 23.20 )
  lineptr = 5;
else if ( 23.20 <= head && head < 23.50 )
  lineptr = 6;
else if ( 23.50 <= head && head < 23.80 )
  lineptr = 7;
else if ( 23.80 <= head && head < 24.10 )
  lineptr = 8;
else if ( 24.10 <= head && head < 24.40 )
  lineptr = 9;
else if ( 24.40 <= head && head < 24.70 )
  lineptr = 10;
else if ( 24.70 <= head && head < 25.00 )
  lineptr = 11;
else if ( 25.00 <= head && head < 25.30 )
  lineptr = 12;
else if ( 25.30 <= head && head < 25.70 )
  lineptr = 13;
else if ( 25.70 <= head && head < 26.00 )
  lineptr = 14;
else if ( 26.00 <= head && head < 26.30 )
  lineptr = 15;
else if ( 26.30 <= head && head < 26.60 )
  lineptr = 16;
else if ( 26.60 <= head && head < 26.90 )
  lineptr = 17;
else if ( 26.90 <= head && head < 27.20 )
  lineptr = 18;
else if ( 27.20 <= head && head <= 27.40 )
  lineptr = 19;

printf("%3d\n",lineptr);
/* Open and read in power points for the specified head */
pp1 = fopen("p1","r");
for(i=1; i<=lineptr; ++i)
  dummy2 = fgets(dummy,260,pp1);
for(i=1; i<=numpoints; ++i)
  fscanf(pp1,"%lf",&power[1][i]);
fclose(pp1);

pp2 = fopen("p2","r");
for(i=1; i<=lineptr; ++i)
```

```
  dummy2 = fgets(dummy,260,pp2);
for(i=1; i<=numpoints; ++i)
  fscanf(pp2,"%lf",&power[2][i]);
fclose(pp2);


/* Open and read in flows at specific power and head levels */
fp1 = fopen("q1","r");

for(i=0; i<=67; ++i)
  for(j=1; j<=21; ++j)
    fscanf(fp1,"%lf",&flowc[i][j]);

fclose(fp1);
typeunit = 1;
calcflow(typeunit,headc);



fp2 = fopen("q2","r");

for(i=0; i<=72; ++i)
  for(j=1; j<=21; ++j)
    fscanf(fp2,"%lf",&flowc[i][j]);

fclose(fp2);
typeunit = 2;
calcflow(typeunit,headc);

}



void calcflow(int typeunit, double headc[22])
{
int equal, isoline, iflow, ipower;

double percent, fraction, flowhi, flowlo;

equal = 0;


/* Find location of isoline */
for(i=21; i>=1; --i)
 {
  if( head == headc[i] )
    {
     isoline = i;
```

```
      equal = 1;
      break;
     }
    else if ( head > headc[i] )
     {
      isoline = i + 1;
      break;
     }
   }

flow[typeunit][1] = 0.0;

if ( equal == 0 )
  {
   percent = (head - headc[isoline-1]) /
           (headc[isoline] - headc[isoline-1] );
   for(i=2; i<=numpoints; ++i)
     {
      ipower = (int)power[typeunit][i];
      if( (power[typeunit][i] - (double)ipower) == 0.0 )
        {
         flow[typeunit][i] = ( flowc[ipower][isoline-1]+ percent*
                  ( flowc[ipower][isoline] -
                    flowc[ipower][isoline-1] )
                 ) * 0.028317;
        }
      else
        {
         flowhi = ( flowc[ipower+1][isoline-1]+ percent*
                  ( flowc[ipower+1][isoline] -
                    flowc[ipower+1][isoline-1] )
                 ) * 0.028317;
         flowlo = ( flowc[ipower][isoline-1]+ percent*
                  ( flowc[ipower][isoline] -
                    flowc[ipower][isoline-1] )
                 ) * 0.028317;
         fraction = power[typeunit][i] - (double)ipower;
         flow[typeunit][i] = flowlo + fraction * ( flowhi - flowlo );
        }
     }
  }
else
  {
   for(i=2; i<=numpoints; ++i)
     {
      ipower = (int)power[typeunit][i];
      if( (power[typeunit][i] - (double)ipower) == 0.0 )
        {
```

```
        flow[typeunit][i] = 0.028317 * flowc[ipower][isoline];
       }
      else
       {
        flowhi = 0.028317 * flowc[ipower+1][isoline];
        flowlo = 0.028317 * flowc[ipower][isoline];
        fraction = power[typeunit][i] - (double)ipower;
        flow[typeunit][i] = flowlo + fraction * ( flowhi - flowlo );
       }
     }
  }

/* Round to 1st decimal place */
for(i=2; i<=numpoints; ++i)
  {
   flow[typeunit][i] *= 10.0;
   iflow = (int) flow[typeunit][i];
   fraction = flow[typeunit][i] - (double)iflow;
   if (fraction < 0.5)
     flow[typeunit][i] = (double)iflow / 10.0;
   else
     flow[typeunit][i] =  ( (double)iflow / 10.0 ) + 0.1;
  }

}



double round1st(double xr)
{
int xi;

double yr;

xr *= 10.0;
xi = (int) xr;
yr = xr - (double) xi;
if( yr < 0.5 )
   xr = (double)xi/10.0;
else
   xr = ( (double)xi/10.0 ) + 0.1;
return(xr);
}



void qflow(void)
{
```

```
/* Determine flow of water through each unit */
k=1;
for(i=1; i<=numunits; ++i)
  {
   q[i] = 0.0;
   if( unittype[i] > 0 )
     {
      for(j=1; j<=numpoints; ++j)
        {
         q[i] += *xpointer[k] * flow[unittype[i]][j];
         ++k;
        }
      q[i] = round1st(q[i]);
     }
  }
}



void active(void)
{
/* Determine active slope segment of each unit */
for(i=1; i<=numunits; ++i)
  {
   segment[i] = 1;
   if( unittype[i] > 0 )
     {
      for(j=numpoints; j>0; --j)
        {
         if( j == numpoints && flow[unittype[i]][j] <= q[i] )
           {
            segment[i] = j;
            break;
           }
         if( flow[unittype[i]][j] <= q[i] )
           {
            segment[i] = j;
            break;
           }
        }
     }
  }
}



void slopealloc(void)
{
```

```
int kunit;

double qremain, allocate, qq, maxslope;

qq = 0.0;

/* Calculate allocated flow */
for(i=1; i<=numunits; ++i)
 {
  qq += q[i];
 }

qremain = qtotal - qq;


/* Allocate remaining water */
while( qremain > 0.0 )
 {
  maxslope = 0.0;
  for(i=1; i<=numunits; ++i)
   {
    if( unittype[i] > 0 )
     {
      if( slope[unittype[i]][segment[i]] > maxslope &&
          segment[i] < numpoints )
       {
        kunit = i;
        maxslope = slope[unittype[i]][segment[i]];
       }
     }
   }

  allocate = flow[unittype[kunit]][segment[kunit]+1] - q[kunit];

  if( allocate <= qremain )
   {
    q[kunit] += allocate;
    ++segment[kunit];
    qremain -= allocate;
   }
  else
   {
    q[kunit] += qremain;
    qremain = 0.0;
   }
 }
}
```

```
void mwpower(void)
{
/* Calculate power */
for(i=1; i<=numunits; ++i)
 {
  mw[i] = 0.0;
  if( unittype[i] > 0 )
   {
    if ( q[i] == flow[unittype[i]][segment[i]] )
     {
      mw[i] = power[unittype[i]][segment[i]];
     }
    else
     {
      mw[i] = power[unittype[i]][segment[i]] +
              slope[unittype[i]][segment[i]] *
              ( q[i] - flow[unittype[i]][segment[i]] ) ;
     }
   }
 }

q[0] = 0.0;
mw[0] = 0.0;
for(i=1; i<=numunits; ++i)
 {
  q[0] += q[i];
  mw[0] += mw[i];
 }
}



void finetune(void)
{
int minunit, maxunit;

double minslope, available, flowdown, maxslope;

maxslope = 1.0;
minslope = 0.0;

while( maxslope > minslope )
 {

  minslope = 1.0e8;
  for(i=1; i<=numunits; ++i)
```

```
{
 if( unittype[i] > 0 && q[i] > 0.0 )
  {
   if( q[i] == flow[unittype[i]][segment[i]] )
    {
     if( slope[unittype[i]][segment[i]-1] < minslope )
      {
       minslope = slope[unittype[i]][segment[i]-1];
       minunit = i;
      }
    }
   else
    {
     if( slope[unittype[i]][segment[i]] < minslope )
      {
       minslope = slope[unittype[i]][segment[i]];
       minunit = i;
      }
    }
  }
}
maxslope = 0.0;
for(i=1; i<=numunits; ++i)
 {
  if( unittype[i] > 0 && i != minunit && segment[i] < numpoints )
   {
    if( slope[unittype[i]][segment[i]] > maxslope )
     {
      maxslope = slope[unittype[i]][segment[i]];
      maxunit = i;
     }
   }
 }


if( maxslope <= minslope)
 {
  maxslope = 0.0;
  for(i=1; i<=numunits; ++i)
   {
    if( unittype[i] > 0 && segment[i] < numpoints )
     {
      if( slope[unittype[i]][segment[i]] > maxslope )
       {
        maxslope = slope[unittype[i]][segment[i]];
        maxunit = i;
       }
     }
```

```
    }
  minslope = 1.0e8;
  for(i=1; i<=numunits; ++i)
    {
     if( unittype[i] > 0 && q[i] > 0.0 && i != maxunit )
       {
        if( q[i] == flow[unittype[i]][segment[i]] )
          {
           if( slope[unittype[i]][segment[i]-1] < minslope )
             {
              minslope = slope[unittype[i]][segment[i]-1];
              minunit = i;
             }
          }
        else
          {
           if( slope[unittype[i]][segment[i]] < minslope )
             {
              minslope = slope[unittype[i]][segment[i]];
              minunit = i;
             }
          }
       }
    }
 }


if( maxslope > minslope )
 {
  if( q[minunit] == flow[unittype[minunit]][segment[minunit]] )
    flowdown = q[minunit] -
               flow[unittype[minunit]][segment[minunit]-1];
  else
    flowdown = q[minunit] -
               flow[unittype[minunit]][segment[minunit]];

  available = flow[unittype[maxunit]][segment[maxunit]+1] -
              q[maxunit];


  if( flowdown < available )
    {
     if( q[minunit] == flow[unittype[minunit]][segment[minunit]] )
       --segment[minunit];
     q[maxunit] += flowdown;
     q[minunit] -= flowdown;
    }
  else
```

```
        {
         if( q[minunit] == flow[unittype[minunit]][segment[minunit]] )
            --segment[minunit];
         q[maxunit] += available;
         q[minunit] -= available;
         ++segment[maxunit];
        }
      }
    }
  }
```